Assembly Language: Function Calls

Goals of this Lecture

• Help you learn:
  • The challenges of supporting functions
    • Providing information for the called function
    • Function arguments and local variables
    • Allowing the calling function to continue where it left off
      • Return address and contents of registers
  • How to use the runtime stack
    • Stack frame: args, local vars, return address, registers
    • Stack pointer: pointing to the current top of the stack
  • How to call functions
    • Call and ret instructions, to call and return from functions
    • Pushing and popping the stack frame
    • Using the base pointer EBP as a reference point
Challenges of Supporting Functions

• Code with a well-defined entry and exit points
  • Call: How does the CPU go to that entry point?
  • Return: How does the CPU go back to the right place, when “right place” depends on who called the function?

• With arguments and local variables
  • How are the arguments passed from the caller?
  • Where should the local variables be stored?

• Providing a return value
  • How is the return value returned to the calling function?

• Without changing variables in other functions
  • How are the values stored in registers protected?

Call and Return Abstractions

• Call a function
  • Jump to the beginning of an arbitrary procedure
  • I.e., jump to the address of the function’s first instruction

• Return from a function
  • Jump to the instruction immediately following the “most-recently-executed” Call instruction

<table>
<thead>
<tr>
<th>P: # Function P</th>
<th>R: # Function R</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>jmp R</td>
<td>jmp Rtn_point1 # Return</td>
</tr>
<tr>
<td># Call R</td>
<td></td>
</tr>
<tr>
<td>Rtn_point1:</td>
<td></td>
</tr>
</tbody>
</table>
Challenge: Where to Return?

P:  # Function P
  ...
  jmp R  # Call R
Rtn_point1:
  ...

Q:  # Function Q
  ...
  jmp R  # Call R
Rtn_point2:
  ...

R:  # Function R
  ...
  jmp ???  # Return

The same function may be called from many places.
What addr should return instruction in R jump to?

Store Return Address in Register?

P:  # Proc P
  movl $Rtn_point1, %eax
  jmp R  # Call R
Rtn_point1:
  ...

Q:  # Proc Q
  movl $Rtn_point2, %eax
  jmp R  # Call R
Rtn_point2:
  ...

R:  # Proc R
  ...
  jmp %eax  # Return

Convention: At Call time, store return address in EAX
Problem: Nested Function Calls

```assembly
P:      # Function P
    movl $Rtn_point1, %eax
    jmp Q      # Call Q
Rtn_point1:
    ...

Q:      # Function Q
    movl $Rtn_point2, %eax
    jmp R      # Call R
Rtn_point2:
    ...
    jmp %eax  # Return

R:      # Function R
    ...
    jmp %eax  # Return
```

• Problem if P calls Q, and Q calls R

• Return address for P to Q call is lost

Solution: Put Return Address on a Stack

• May need to store many return addresses
  • The number of nested functions is not known in advance
  • A return address must be saved for as long as the function invocation continues

• Addresses used in reverse order
  • E.g., function P calls Q, which then calls R
  • Then R returns to Q which then returns to P

• So, need last-in-first-out data structure: A Stack
  • Calling function pushes return address on the stack
  • … and called function pops return address off the stack
Arguments to the Function

- Calling function needs to pass arguments
  - Cannot simply put arguments in a specific register
  - Because function calls may be nested
- So, put the arguments on the stack, too!
  - Calling function pushes arguments on the stack
  - Called function loads/stores them on the stack

```c
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

```c
int foo(void) {
    return add3(3, 4, 5);
}
```

Local Variables

- Local variables: called function has local variables
  - Short-lived, so don’t need a permanent location in memory
  - Size known in advance, so don’t need to allocate on the heap
- So, the function just uses the top of the stack
  - Store local variables on the top of the stack
  - The local variables disappear after the function returns

```c
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

```c
int foo(void) {
    return add3(3, 4, 5);
}
```
Registers

- Registers
  - Small, fast memory (e.g., directly on the CPU chip)
  - Used as temporary storage for computations

- Cannot have separate registers per function
  - Could have arbitrary number of nested functions
  - Want to allow each function to use all the registers

- Could write all registers out to memory
  - E.g., save values corresponding to program variables
    - Possible, but a bit of a pain…
  - E.g., find someplace to stash intermediate results
    - Where would we put them?

- Instead, save the registers on the stack, too

Stack Frames

- Use stack for all temporary data related to each active function invocation

  - Return address
  - Input parameters
  - Local variables of function
  - Saving registers across invocations

- Stack has one Stack Frame per active function invocation
High-Level Picture

main begins executing

%ESP

Bottom

main’s Stack Frame

High-Level Picture

main begins executing
main calls P

%ESP

Bottom

P’s Stack Frame
main’s Stack Frame
High-Level Picture

main begins executing
main calls P
P calls Q

%ESP

Bottom
High-Level Picture

main begins executing
main calls P
P calls Q
Q calls P
P returns

High-Level Picture

main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R

%ESP
0

main’s Stack Frame

main’s Stack Frame

Q’s Stack Frame

P’s Stack Frame

%ESP
0

main’s Stack Frame

main’s Stack Frame

Q’s Stack Frame

P’s Stack Frame

R’s Stack Frame

Bottom

Bottom
High-Level Picture

main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns

High-Level Picture

main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
Q returns
High-Level Picture

- main begins executing
- main calls P
- P calls Q
- Q calls P
- P returns
- Q calls R
- R returns
- Q returns
- P returns

High-Level Picture

- main begins executing
- main calls P
- P calls Q
- Q calls P
- P returns
- Q calls R
- R returns
- Q returns
- P returns
- main returns
Function Call Details

- **Call and Return instructions**
  - Call: push EIP on the stack, and jump to function
  - Return: pop from stack into the EIP to go back

- **Argument passing between procedures**
  - Calling function pushes arguments on to the stack
  - Called function reads/writes on the stack

- **Local variables**
  - Called function creates and manipulates on the stack

- **Register saving conventions**
  - Either calling or called function saves all of the registers

Call and Return Instructions

| Instruction | Effective Operations                                      | \%ESP before Call
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
<td>%ESP</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
<td></td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
<td></td>
</tr>
<tr>
<td>call addr</td>
<td>pushl %eip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>pop %eip</td>
<td></td>
</tr>
</tbody>
</table>

Note: can’t really access EIP directly, but this is implicitly what call and ret are doing.
Call and Return Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
</table>
| pushl src   | subl $4, %esp  
             movl src, (%esp) |
| popl dest   | movl (%esp), dest  
             addl $4, %esp |
| call addr   | pushl %eip  
             jmp addr |
| ret         | pop %eip |

Return instruction assumes that the return address is at the top of the stack.
Call and Return Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushl %eip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
<tr>
<td>ret</td>
<td>pop %eip</td>
</tr>
</tbody>
</table>

Return instruction assumes that the return address is at the top of the stack.

Input Parameters

- Caller pushes input parameters before executing the Call instruction.
- Parameters are pushed in the reverse order:
  - Push Nth argument first
  - Push 1st argument last
  - So that first argument is at the top of the stack at the time of the Call.
Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push N\textsuperscript{th} argument first
  - Push 1\textsuperscript{st} argument last
  - So that first argument is at top of the stack at the time of the Call

\[ \text{%ESP before Call} \]

\begin{align*}
\text{Arg } N \\
\text{Arg } 1 \\
\text{Arg } \ldots \\
\end{align*}

Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push N\textsuperscript{th} argument first
  - Push 1\textsuperscript{st} argument last
  - So that first argument is at top of the stack at the time of the Call

\[ \text{%ESP after Call} \]

\begin{align*}
\text{Old EIP} \\
\text{Arg } 1 \\
\text{Arg } \ldots \\
\text{Arg } N \\
\end{align*}

Called function can address arguments relative to ESP: Arg 1 as 4(%esp)

Why is the EIP put on after the arguments?
Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that first argument is at top of the stack at the time of the Call

After the function call is finished, the caller pops the pushed arguments from the stack
Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push N\textsuperscript{th} argument first
  - Push 1\textsuperscript{st} argument last
  - So that first argument is at top of the stack at the time of the Call

After the function call is finished, the caller pops the pushed arguments from the stack.

Base Pointer: EBP

- As Callee executes, ESP may change
  - E.g., preparing to call another function
- Use EBP as fixed reference point
  - E.g., to access arguments and other local variables
- Need to save old value of EBP
  - Before overwriting EBP register
- Callee begins by executing “prolog”
  - \texttt{pushl \%ebp}
  - \texttt{movl \%esp, \%ebp}
  - \texttt{\%EBP}
**Base Pointer: EBP**

- As Callee executes, ESP may change
  - E.g., preparing to call another function
- Use EBP as fixed reference point
  - E.g., to access arguments and other local variables
- Need to save old value of EBP
  - Before overwriting EBP register
- Callee begins by executing “epilog”
  - `pushl %ebp`
  - `movl %esp, %ebp`
- Regardless of ESP, Callee can address Arg 1 as 8(%ebp)

---

**Base Pointer: EBP**

- Before returning, Callee must restore EBP to its old value
- Executes
  - `movl %ebp, %esp`
  - `popl %ebp`
  - `ret`
Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes
  
  ```
  movl %ebp, %esp
  popl %ebp
  ret
  ```

Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes
  
  ```
  movl %ebp, %esp
  popl %ebp
  ret
  ```
Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes
  \[
  \text{movl} \ %\text{ebp}, \ %\text{esp} \\
  \text{popl} \ %\text{ebp} \\
  \text{ret}
  \]

Allocation for Local Variables

- Local variables of the Callee are also allocated on the stack
- Allocation done by moving the stack pointer
- Example: allocate two integers
  - subl $4, %esp
  - subl $4, %esp
  - (or equivalently, subl $8, %esp)
- Reference local variables using the base pointer
  - -4(%ebp)
  - -8(%ebp)
Use of Registers

• Problem: Called function may use a register that the calling function is also using
  • When called function returns control to calling function, old register contents may be lost
  • Calling function cannot continue where it left off
• Solution: save the registers on the stack
  • Someone must save old register contents
  • Someone must later restore the register contents

• Need a convention for who saves and restores which registers

GCC/Linux Convention

• Caller-save registers
  • %eax, %edx, %ecx
  • Save on stack (if necessary) prior to calling
• Callee-save registers
  • %ebx, %esi, %edi
  • Old values saved on stack prior to using, and restored later
• %esp, %ebp handled as described earlier
• Return value is passed from Callee to Caller in %eax
A Simple Example

```c
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

int foo(void)
{
    return add3(3, 4, 5);
}
```

In general, one may need to push callee-save registers onto the stack.

Add the three arguments:
- `movl 8(%ebp), %eax`
- `addl 12(%ebp), %eax`
- `addl 16(%ebp), %eax`

Put the sum into `d`:
- `movl %eax, ~4(%ebp)`

Return value is already in eax.

In general, one may need to pop callee-save registers.

Restore old ebp, discard stack frame:
- `movl %ebp, %esp`
- `popl %ebp`

Return:
- `ret`
A Simple Example

```c
int foo(void) {
    return add3( 3, 4, 5 );
}
```

```c
int add3(int a, int b, int c) {
    return a + b + c;
}
```

```c
foo:
    # Save old ebp, and set-up
    pushl %ebp
    movl %esp, %ebp
    pushl %ebp
    movl %esp, %ebp
    # No local variables
    # No need to save callee-save
    # registers as we
    # don't use any registers
    call add3
    addl $12, %esp
    # Return value is already in eax
    movl %ebp, %esp
    popl %ebp
    # Return
    ret
```

Conclusion

- **Invoking a function**
  - Call: call the function
  - Ret: return from the instruction

- **Stack Frame for a function invocation includes**
  - Return address,
  - Procedure arguments,
  - Local variables, and
  - Saved registers

- **Base pointer EBP**
  - Fixed reference point in the Stack Frame
  - Useful for referencing arguments and local variables